NPS-EC-92-007

NAVAL POSTGRADUATE SCHOOL

Monterey, California

REAL-TIME EXECUTION CONTROL OF
TASK-LEVEL DATA-FLOW GRAPHS
USING A COMPILE-TIME APPROACH

Shridhar B. Shukla
Brian Little
Amr Zaky

April 1992

Prepared for: Naval Sea Systems Command
Washington, DC

Naval Postgraduate School
Monterey, California 93943-5000

Rear Admiral R.W. West, Jr.
Superintendent

H. Shull
Provost

This report was prepared for and funded by the Naval Sea Systems Command, PMS 412, Washington, D.C.

This report was prepared by:

REPORT DOCUMENTATION PAGE

REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
DECLASSIFICATION/DOWNGRADING SCHEDULE	
PERFORMING ORGANIZATION REPORT NUMBER(S) NPS-EC-92-007	5. MONITORING ORGANIZATION REPORT NUMBER(S)
NAME OF PERFORMING ORGANIZATION Dept. of Elect. & Comp. Eng. Naval Postgraduate School 6b. OFFICE SYMBOL (*if applicable*) EC/Sh	7a. NAME OF MONITORING ORGANIZATION TRW
ADDRESS (*City, State, and ZIP Code*) Monterey, CA 93943-5004	7b. ADDRESS (*City, State, and ZIP Code*) Washington, DC 20362
NAME OF FUNDING/SPONSORING ORGANIZATION NAVSEA 8b. OFFICE SYMBOL (*if applicable*) PMS 412	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
ADDRESS (*City, State, and ZIP Code*) Washington, DC 20362	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

TITLE (*Include Security Classification*)

Real-time Execution Control of Task-level Data-flow Graphs Using A Compile-time Approach

PERSONAL AUTHOR(S)

Shridhar B. Shukla, Brain Little, and Amr Zaky

TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (*Year, Month, Day*)	15. PAGE COUNT
Technical Report	FROM 10/1/90 TO 3/31/91	April 1991	29

SUPPLEMENTARY NOTATION

The views expressed in this report are those of the author and do not reflect the cial policy or position of the Department of Defense or the United States Government.

COSATI CODES		
IELD	GROUP	SUB--GROUP

18. SUBJECT TERMS (*Continue on reverse if necessary and identify by block number*)

Compile-time, Data-flow, Graph Restructuring, Real-time, Run-time, Scheduling, Signal Processing, Throughput, Task-level

ABSTRACT (*Continue on reverse if necessary and identify by block number*)

Efficient data-flow implementation requires fast run-time mechanisms to detect and dispatch schedulable tasks. However, the rent non-determinism in data-flow executions and the requirement of fast, and therefore, simple run-time mechanisms necessitate pile-time support to improve performance. In particular, for data-flow execution of applications, such as signal processing which characterized by periodically received data, compile-time support can be used to control the run-time behavior to improve the ictability and efficiency. In this report, a compile-time technique that supports a simple run-time mechanism to improve ughput and predictability for a task-level data-flow programming model is described. This technique, called the *revolving ader* analysis, restructures the application, described by a task-level data-flow graph. The restructuring is based on wrapping projected data-flow execution trace on the curved surface of a cylinder whose area depends upon the number of processors the sum of the task execution times. The behavior of the restructured graph is shown to be more predictable under the e run-time mechanism than that of the old graph. Results on the performance improvement for two typical signal processing ications, *viz.*, a correlator and a fast Fourier Transform, are presented. The potential of this approach in determining the mal granularity for an application is also described.

DISTRIBUTION/AVAILABILITY OF ABSTRACT ☐ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
NAME OF RESPONSIBLE INDIVIDUAL Shridhar B. Shukla	22b. TELEPHONE (*Include Area Code*) (408) 646-2764 22c. OFFICE SYMBOL EC/Sh

Real-time Execution Control of Task-level Data-flow Graphs Using A Compile-time Approach[1]

by

Shridhar Shukla and Brian Little[2]
Code EC/Sh, Dept. of Elect. & Computer Engineering
Naval Postgraduate School
Monterey, CA 93943-5000
Tel: (408) 646-2764 Fax: (408) 646-2760
E-mail: shukla@ece.nps.navy.mil
and
Amr Zaky
Code CS/Za, Department of Computer Science
Naval Postgraduate School
Monterey, CA 93943-5000
Tel: (408) 646-2693 Fax: (408) 646-2814
E-mail: zaky@cs.nps.navy.mil

[1] This research has been supported in part by PMS 412, Naval Sea Systems Command, Department of Navy, Washington DC, 20362

[2] Currently at the Naval Submarine School, Groton, CT

This is technical report NPS-EC-92-007. It supercedes NPS-EC-92-002.

Abstract

Efficient data-flow implementation requires fast run-time mechanisms to detect and dispatch schedulable tasks. However, the inherent non-determinism in data-flow executions and the requirement of fast, and therefore, simple run-time mechanisms necessitate compile-time support to improve performance. In particular, for data-flow execution of applications, such as signal processing which are characterized by periodically received data, compile-time support can be used to control the run-time behavior to improve the predictability and efficiency. In this report, a compile-time technique that supports a simple run-time mechanism to improve throughput and predictability for a task-level data-flow programming model is described. This technique, called the *revolving cylinder* analysis, restructures the application, described by a task-level data-flow graph. The restructuring is based on wrapping the projected data-flow execution trace on the curved surface of a cylinder whose area depends upon the number of processors and the sum of the task execution times. The behavior of the restructured graph is shown to be more predictable under the same run-time mechanism than that of the old graph. Results on the performance improvement for two typical signal processing applications, *viz.*, a correlator and a fast Fourier Transform, are presented. The potential of this approach in determining the optimal granularity for an application is also described.

1 Introduction

Data-flow graphs not only describe the dependencies between different parts of the computation required in an application, but also provide built-in scheduling and synchronization. For example, on a hypothetical system with no communication cost and an unlimited number of processors, nodes can synchronize by sending data and a node can be scheduled as soon as all the required data is present at its input. Due to the generality of this representation, it can be used to specify parallelism at the instruction level [Bro87, SFP83] as well as at the task level [LM87]. The theoretical foundation for the consistency of such representations has been well studied [KM66, Lee91]. In practical implementations of this paradigm, the machine must provide mechanisms to manage the data that flows through the graph and to capture the intrinsic scheduling and synchronization. These mechanisms, typically operating at run-time, result in overheads that lead to suboptimal performance. The amount of overhead depends critically on the granularity of the parallelism expressed by the graph and on whether the computations have conditionals and recursion. A direct implementation in hardware of the data-flow paradigm for general applications results in unmanageable overheads [GKW85, Bro87].

However, for classes of applications, such as signal processing, data-flow can be managed very efficiently to obtain significant performance improvement. The two properties of these applications that make this possible are availability of *a priori* knowledge of the amount of data produced and consumed and negligible use of conditionals and recursion. When the amounts of data produced and consumed by the nodes of a data-flow graph are known exactly, the applications are called *synchronous data-flow* applications [LM87]. When the data arrives periodically, they have been classified as *pipelined function-parallel computations* [KCN90].

Any data-flow implementation must perform buffering and fetching of data, allocation of graph nodes to processors, their ordering on each, and the exact times at which they are scheduled. If all the related decisions are done at run-time, the efficiency of the implementation suffers. The overheads can be reduced effectively by using the node and arc attributes of the data-flow graph at compile-time to simplify the run-time management.

Based on which decisions are made at compile-time and which ones are made at run-time, data-flow implementations can be classified over a spectrum that ranges from *fully-static* to *fully-dynamic* [LB90]. While dynamic implementations have more overhead, they are more flexible and are easier to implement. They also degrade *gracefully* in the even of individual processor malfunction. On the other hand, static implementations are more efficient and lead to predictable performance which is crucial to real-time systems. However, they are difficult to realize, are inflexible, and do not degrade gracefully. Their effectiveness is determined by how accurately the computational requirements of the application are known. This is typically a difficult problem and its solution of using the worst-case estimate can result in large inefficiencies. Therefore, real-time systems must strike a careful balance between the compile-time effort and run-time complexity to get the desired and guaranteed performance.

In real-time signal processing applications, the trade-offs between compile-time and run-time has an additional dimension because of the periodic arrival of data. When external data arrives periodically, the intrinsic non-determinism of data-flow execution results in unpredictable program behavior. As a result, processed data arrives unpredictably leading to the possibility of intolerable delays and insufficient buffer space, especially under high loads.

The focus of this work is on compile-time mechanisms for controlling data-flow implemented using a simple run-time mechanism for real-time signal processing applications. We present a technique in which, instead of generating information, such as schedules, to control allocation or ordering on processors at run-time, a new data-flow graph is obtained as a result of the compile-time analysis. The behavior of this new graph is more predictable under the same run-time mechanism than that of the old graph. Section 2 describes a model for task-level data-flow processing and illustrates the problems associated with fully dynamic data-flow execution of real-time signal processing applications. Section 3 describes the proposed approach and presents the graph restructuring algorithm. Section 4 describes the effectiveness of this approach on two applications using the results of a simulation. Finally, in Section 5, the potential of graph restructuring and how this approach can be developed further is described.

2 A Model for Task-level Data-flow in Signal Processing

Figure 1 shows the architectural model under consideration for task-level data-flow. This model closely resembles the AN/UYS-2 parallel signal processor developed by the US Navy [Ric90]. The model has four basic types of elements, *viz.*, the processors (P), memory modules (M), scheduler (SCH), and the interconnection network. The processors execute individual nodes of the data-flow graph. Each processor has a local memory in which data on all the input queues as well as the instruction stream corresponding to the node are first fetched. All input and output queues of the graph [1] are stored in the memory modules. The memory modules monitor the state of these queues, *i. e.*, whether there is space for additional data, the amount of data has gone above or below certain predetermined threshold and capacity levels. Changes in the status of a queue are sent to the scheduler. This information is used by the scheduler to make run-time decisions. Memory modules also store the instruction streams for all the nodes in the graph. The instruction stream and data are moved between the processors and the memory modules across the interconnection network. The scheduler itself is a simple run-time dispatcher that matches the free processors in the free processor list (FPL) with the ready nodes in the ready node list (RNL). The operation of this architecture is briefly described below.

2.1 Data-flow Execution

Applications are specified as data-flow graphs which are *directed, acyclic graph* with nodes representing large grain computations [2] chosen from a library of signal processing functions. The edges of a graph represent queues which receive data from the source node and supply data to the destination node. Each queue is allocated to a memory module for storage which maintains its current size and the remaining capacity. As data arrives on all the input queues of a node, the threshold values associated with each queue is eventually exceeded. Threshold refers to the minimum number of data items that must be present in a queue for its destination to become ready. A node is *ready* for execution when two conditions are satisfied.

[1] Unless otherwise mentioned, the term *graph* always refers to a data-flow graph in the rest of the report.
[2] Each node can be a complex program.

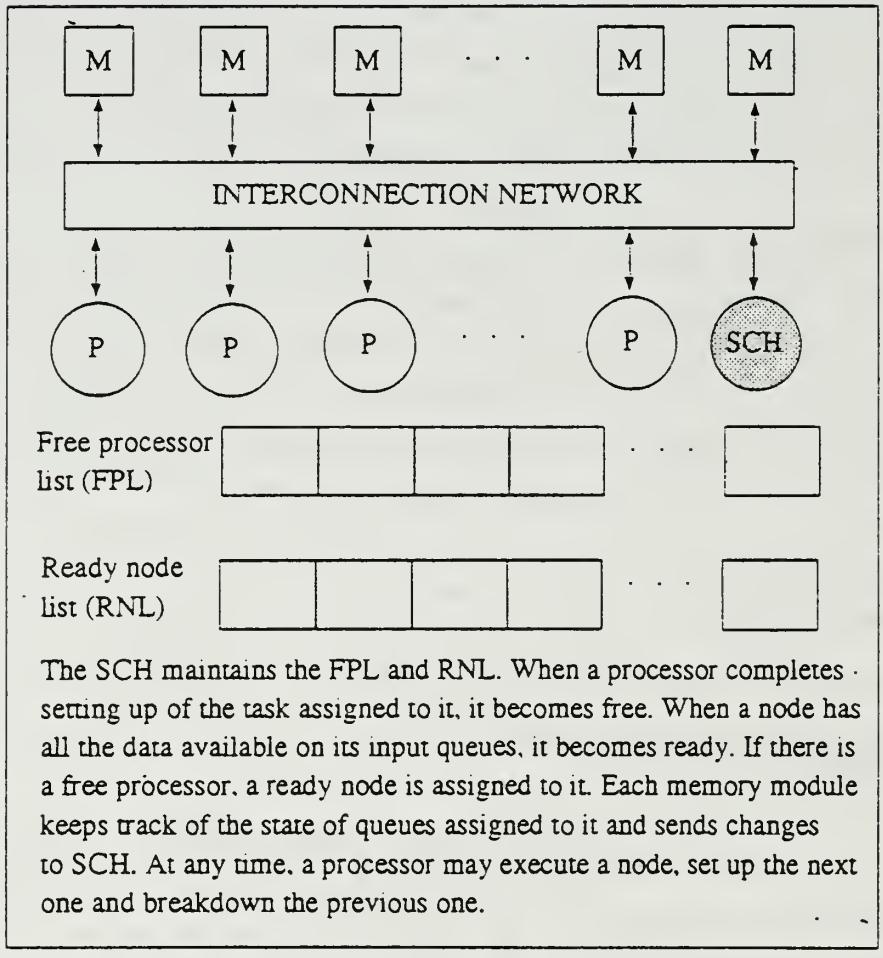

Figure 1: Model of A Parallel Task-level Data-flow Processor

All its incoming queues exceed their thresholds and all its output queues must be under their capacity values. All memory modules communicate the events of threshold/capacity crossing to the scheduler which determines if a node is *ready*. Initially all processors are on the FPL and the scheduler assigns them to nodes on the RNL. When a node is assigned to a processor, it fetches the data and the instruction stream corresponding to the node from appropriate memory module. When the entire instruction stream and queue data have been fetched, the *setup* of the node is complete. A processor communicates this event to the scheduler to get itself placed on the FPL so that the next node may start getting set up. Thus, the node already *setup* begins execution while the next node gets *setup* with the restriction that a processor may have only one node *setup* and *pending to execute* at any time. The data generated by the execution is first stored locally. Upon completion, a processor transfers the data to appropriate memory module storing the output queues in what is referred to as the *breakdown* phase. Thus, any node goes through three phases at a processor, *viz.*, *setup*, *execution*, and *breakdown*. Since their functions are independent and the *set-up/breakdown* operations may require time comparable to the execution time, these operations can be overlapped by providing independent functional units for execution unit and data movement unit within a processor.

Upon arrival of sufficient data at the nodes which receive data only from the external world, an instance of the graph is started and its execution proceeds according to the data-flow principle. As a result of the data-flow execution, which corresponds to asynchronous task-level pipelining, several instances of the graph are active simultaneously. Aside from the requirement that the required throughput must be met by the machine, real-time performance may require that all instances of the graph should complete in the same amount of time. Between the completion of the setup of a node at a processor and the actual start of its execution, there may be a delay because the execution unit at a processor has not completed the previous node. This delay, that may be experienced by a ready node, is in addition to the delay it may experience waiting on the RNL. Both delays result in an increase in the latency of the graph execution. On the other hand, an execution unit may have to wait for the setup completion of the next node assigned to it after it completes its current node. If this happens, execution cycles are lost and the machine throughput degrades.

To maximize throughput, all execution units must run all the time, and therefore, each processor must have some node set up for execution at the time it finishes the previous node

computation. Since the scheduler is a simple run-time dispatcher that matches RNL nodes to free processors, the delays described above depend upon the application execution profile. This profile depends upon the data rate, the *spatial* and *temporal* parallelism in the graph, the number of processors, the number of memory modules. and the allocation of queues to memory modules. Since task-level parallelism is being considered, performance can be improved significantly if setup and breakdown cost can be minimized. One method to reduce this cost is to *chain* successive nodes together and execute them on a single processor one after the other. This results in saving the breakdown cost for the first node and setup cost for the second node.

2.2 Unpredictability in Program Behavior

In real-time environments, the ability to predict the program performance is critical for efficient allocation of resources such as memory modules, processors, and queue sizes. However, the *first-come-first-served* (FCFS) assignment of processors to ready nodes in the above data-flow model is intrinsically non-deterministic. This non-determinism manifests itself as degraded performance in two ways, *viz.*. irregular execution patterns and interference at the memory modules.

When data arrives periodically, the unpredictable execution patterns arise due to the absence of direct control over execution of nodes that depend only upon the receipt of data from the external world. If the output queue capacities for these nodes were unlimited, they would execute at a rate that matches the input arrival and is independent of the rate at which other nodes execute. In the presence of finite queue sizes, they execute at the input rate until the output queues get filled; and then. stall until nodes down the graph create space in the queues by consuming data. This leads to the individual graph instances not being executed in a uniform manner. This is undesirable in real-time scenarios. In addition. the machine throughput will degrade because the memory access patterns may be such that there is interference at the memory modules while setting up and breaking down nodes.

This problem of controlled dat-flow execution has been addresses in different contexts before. For example, in [SMS90], input control has been applied to real-time execution of of graphs on multicomputers. In order to achieve predictability, a custom operating environment called

AMOS has been developed. In [SA91], similar unpredictability has been observed due to the FCFS nature of self-routing of messages in a multicomputer network. The solution proposed therein is a sequence of explicit scheduling of the communication resources. In the following section, a framework is presented that introduces additional dependencies in the graph based on the technique of *revolving cylinder* analysis. While only the problem of controlling execution is addressed in this report, the technique is general enough to be addressed to other problems such as reducing the memory contention and determining the optimal granularity for a given machine configuration.

3 Graph Restructuring Using Revolving Cylinder Analysis

The important resources to be assigned in the model of Fig. 1 are processors and memory modules. We do not address the problem of allocating data queues to memory modules so that memory contention is minimized in this report. The scheduler assigns processor resources on a FCFS basis. The key idea in restructuring based on RC analysis is that inserting dependencies in the graph can produce a graph with better performance. This idea can be traced back to algorithms for overlapping complex operations on pipelined processors [RGP82]. This restructuring selectively changes the conditions when a node will enter the RNL; however, choosing the processor to schedule it on is left to the run-time dispatcher. This enables the actual scheduling to remain dynamic keeping the run-time overhead low.

3.1 Revolving Cylinder (RC) Analysis

Given a graph as in Fig. 2, it is possible to systematically determine whether it can be mapped on a certain number of processors while satisfying the required data rates. For simplicity, we neglect the breakdown and setup times of each node. It can be proved that the graph could be scheduled (ignoring overheads) such that the consecutive graph instances are separated - on the average - t steps away from each other, where t is equal to the total execution time of the PGM divided by the number of processors. This corresponds to the maximal average throughput since the processors will be fully utilized. Thus, for the graph

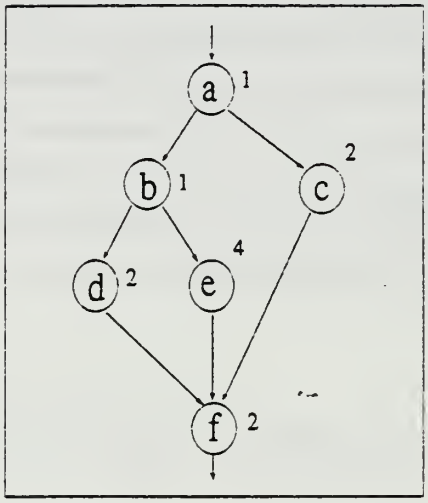

Figure 2: A Simple Data-flow Graph

of Fig. 2, in which the execution times are shown alongside the nodes, a new instantiation could be started every $6(=\frac{12}{2})$ cycles when 2 processors are used. We assume, for simplicity of explanation, that data arrives at this exact rate, although it is not a necessary condition for the algorithms discussed later. The graph of Fig. 2 can be modified by inserting delays as shown in Fig. 3. A schedule for an instance of the modified PGM is shown in Table 1. Another instance of the modified graph can be overlapped with the first instance after six clock cycles, and so on. The idea of adding delays to improve overall throughput at the expense of latency for a single instance has been discussed in the context of hardware pipelining in [Kog81].

For this graph, except for the first 6 processor cycles, which represent a transient, every subsequent group of six consecutive cycles could be summarized by the schedule in Table 2. Table 2 could be derived from Table 1 as follows. Assume that there is a cylinder whose circumference is the intended length of Table 3.1 (6 in this example) and whose height is the number of processors, 2 in this example. Hence, Table 2 (or any table of size 6 by 2) could be wrapped around the cylinder such that its end meets its beginning. The line on the surface of the cylinder that separates the end from the beginning has the effect of a *divide-by-C* counter, where C is the circumference, every time it is crossed to enter the beginning from the end. Now, the first six cycles of Table 1 could be wrapped around the cylinder, then the

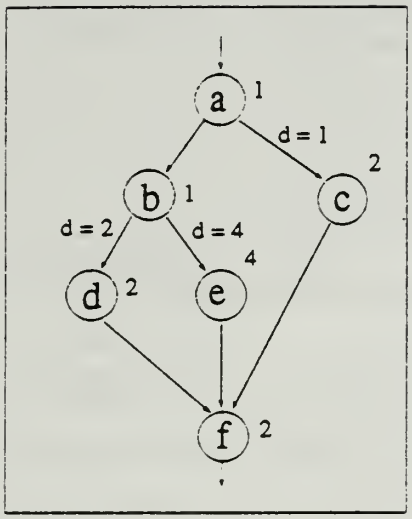

Figure 3: Example graph with Delays Inserted

Table 1: A Schedule for One Instance of the Example Graph with Delays

Cycle #	AP_1	AP_2
1	*a*	
2	*b*	
3	*c*	
4	*c*	
5	*d*	*e*
6	*d*	*e*
7		*e*
8		*e*
9		*f*
10		*f*

Table 2: Compact Representation of RC Assignment

Cycle # ($i \geq 1$)	AP_1	AP_2
$6i-5$	a_i	e_{i-1}
$6i-4$	b_i	e_{i-1}
$6i-3$	c_i	f_{i-1}
$6i-2$	c_i	f_{i-1}
$6i-1$	d_i	e_i
$6i$	d_i	e_i

second six cycles (and generally the process is continued until the table is fully wrapped). The choice of delays in the graph of Fig. 3 and the circumference of the cylinder is such that when Table 1 is wrapped around the cylinder, no node is going to lay over another node. Hence, the cylinder mapping is conflict-free. One minor complication to the above procedure is to assign indices to the nodes on the surface of the cylinder to match those in Table 2. This is established by initially giving index i to all nodes and subtracting from the index of a node the number of revolutions taken around the cylinder before it is assigned its processor cycle(s). This is done to preserve the correctness of the graph, since for our example, e_1 cannot be started at the same time as a_1 is, yet e_0 can be.

Figure 4 illustrates how the entries of the cylinder are indexed. It illustrates that a node can start and continue across the surface boundary. The execution of a node, X, can be split in two parts of length a and b as shown. The upper part has index $i-1$ because, even though it is a continuation of the lower part, the index has decreased by one as we go around the surface once.

The above procedure assumes that the cylinder's circumference and the modified graph with delays on its edges are given. The circumference of the cylinder is equal to the length of Table 2 and is equal to the smallest integer such that a new graph instance could be separated from the previous one. On the other hand, the delays on the edges are not part of the original problem and were used for the sake of clarity. In reality, the delays are not needed to be known *a priori*. A scheduling algorithm could be devised to take the graph in Fig. 2 and obtain the cylindrical assignment of Table 2 without using the information given in Fig. 3 or Table 1. This algorithm is given Fig. 5.

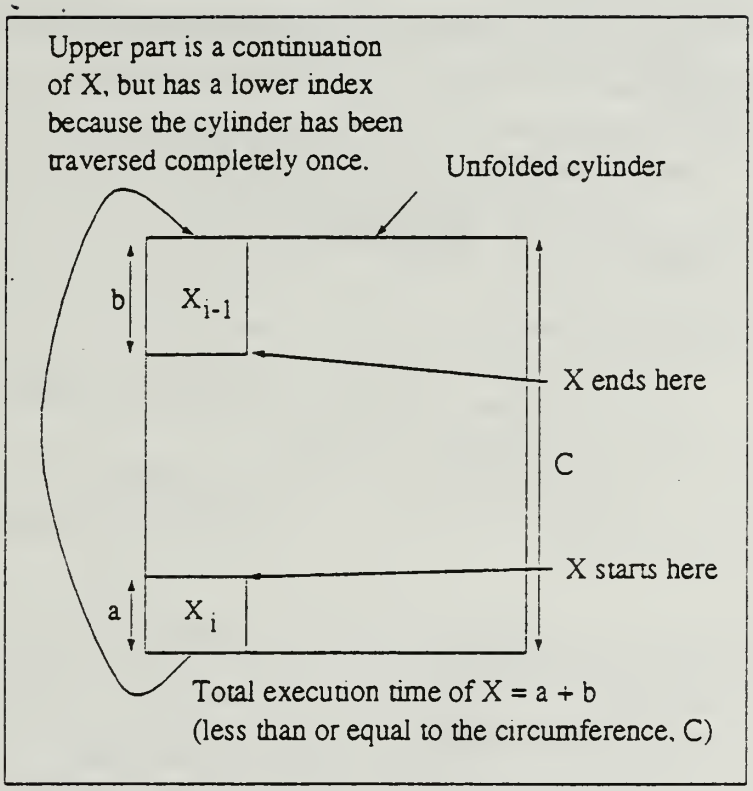

Figure 4: Illustration of Index Assignment

```
procedure Assign_RC (G, p); /*G is directed acyclic graph*/
    /*p is the number of processors*/
    q ← topological sort (G); /*O(e), q is a queue*/
    for all nodes n_i
        est(n_i) ← 0; /* est is the earliest starting time of a node*/
    circumference ← 0
    for all nodes n_i
        circumference ← circumference + w(n_i)
            /*w(n_i) is the size of node n_i*/
    circumference ← ⌈circumference / p⌉;

    while q is not empty
        temp ← remove_top(q);
        t ← schedule_node(temp, est(temp), cylinder) ;
        for all descendents of temp
            est(descendent) ← max(est(descendent), t + w(temp));
    end(while)


procedure schedule_node(temp, t, cylinder)
    scheduled ← false:
    while not scheduled
        try to place temp on cylinder surface slot
            starting at t' = t mod circumference
        if inserted
            scheduled ← true;
        else t' ← (t' + 1)mod circumference:
    end(while);
    return t';
```

Figure 5: An Algorithm to Perform RC Assignment

The algorithm of Fig. 5 is guaranteed if all the node execution times are equal, otherwise there is a chance that it can fail. However, this drawback can be easily remedied as follows. Assign_RC can be used to schedule k copies of the graph, G, on a cylinder whose circumference is $\frac{k}{p} \sum node\ weights$ and k is iteratively increased until it works. The case of $k = p$ is guaranteed to work since the circumference then equals the sum of node weights: however, it is desirable to have k as small as possible.

It should be noted that different schedules which sustain the maximal load could be obtained for any graph. Any assignment of nodes on the surface of the cylinder such that no node is preempted, and no two nodes are mapped to the same square is valid. The availability of multiple schedules which could sustain the same throughput has an important advantage with respect to determining the optimal granularity. For example, nodes can be grouped together on the surface of the cylinder so as to introduce optimizations to minimize the loss of processor cycles due to such overheads as setup and breakdown times or to minimize the interference due to memory accesses.

3.2 Graph Restructuring

Since the run-time mechanism of the scheduler is fixed, any execution sequence enforcement must be accomplished by compile-time techniques. The dashed lines in Fig. 6 show the graph of Fig. 2 with the additional data-dependencies used to enforce RC assignment at run-time. Each dashed line represents a queue of tokens generated by the source and absorbed by the destination. Each source generates a single token when it completes execution. The 2-tuple associated with each indicates the threshold and consume amounts for the control token flow on these arcs. The threshold amount refers to the number of tokens that must be present on the arc for its destination node to be eligible for execution. The consume amount refers to the number of tokens removed from the arc when it executes once. Thus, the arc from b to c forces node c to go on the RL *only after* b has completed. Given such restructuring, the setup and breakdown times for arcs (a, b), (b, d) (a, c) and (e, f) are saved by employing chaining as described at the end of subsection 2.1. It is assumed that implementing the control-token queues has an overhead cost that is negligible with respect to the cost of implementing data queues. It is further assumed that a node can be declared ready if all the data queues have crossed their thresholds, thus enabling a processor to begin its setup by fetching the

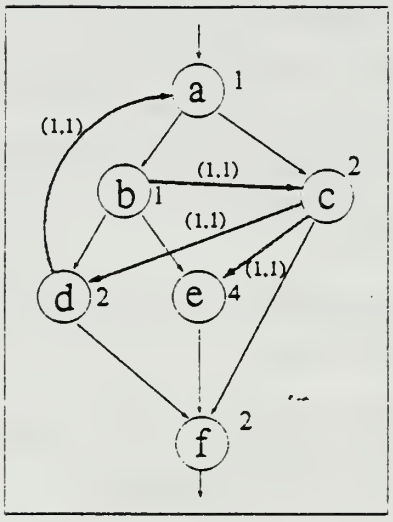

Figure 6: Restructured PGM Graph

instructions and data associated with it although the control queues have not reached the threshold. Thus, the control token queues simply control the execution sequence on each processor. The algorithm to restructure the graph is given in Fig. 7.

The restructuring of the graph in the example above is not unique. Since there are several ways of filling the table, there is a corresponding set of additional arcs. Even for a single assignment, there exist several sets of additional dependencies. This introduces the problem of selecting the best assignment and a suitable set of arcs associated with it for an arbitrary graph. The criteria that can be used for such selection are minimization of the contention for resources or the number of additional arcs introduced.

3.3 Advantages of RC Analysis

There are several advantages of such node-AP assignment if a compile-time technique can be found to enforce it on the scheduler run-time mechanism. Compile-time analysis of whether the machine will meet the required data rate becomes easy. Data-flow execution can be carried out in a controlled manner, thus improving predictability. Since the nodes are scheduled relative to each other at compile-time, it becomes possible to take into consideration

```
procedure Restructure_graph (cylinder, circumference, G);
    /*n_r, n_s are nodes of graph, G*/
    for all nodes, n_r
        check index i of n_r
        find the latest node, n_s, that ends
            before n_r starts on the cylinder
        check index j of n_s
        /*if n_r starts at the top of the cylinder, the latest*/
        /*node ends at the bottom of the cylinder.*/
        /*In this case, j should be decremented by one*/
        introduce a synchronization arc from n_s to n_r
        if i ≥ j
            put i − j initial tokens on the arc
            set threshold = 1, consume = 1
        else if i < j
            put 0 initial tokens on the arc
            threshold = j − i, consume = 1
    end(for)
```

Figure 7: Algorithm to Restructure the graph

the granularity of the graph. *Chaining* has been mentioned as a technique to minimize the cost of *setup/breakdown* of each node. However, unrestrained use of chaining decreases the amount of parallelism in the application. RC analysis offers a systematic method to determine the nodes to be chained and the resulting performance gain. For example, although it is possible to assign nodes in the above example in several ways, the assignment shown enables chaining nodes a, b, c, and d together and chaining e and f together to minimize the setup and breakdown overheads. Thus, such an assignment can potentially take into account the overhead costs while mapping the cylinder. Once it has been determined which nodes are to be chained, the data queues can be allocated to memory modules so that contention is minimized.

4 The Effectiveness of Graph Restructuring

This section presents simulation results on the usefulness of graph restructuring for controlled data-flow execution of two typical signal processing applications. The *correlator* graph is a simple application while the fast Fourier Transform is a communication intensive graph. The predictability is modeled as the non-uniformity in the interval between two successive graph instance completions. This non-uniformity is observed as the interval between successive input data sets is varied up to the maximum possible on an ideal machine for the given graph. As mentioned previously, the minimum input data period is obtained by summing the task execution times and dividing by the number of processors. The plots in the next section are obtained by plotting the input data periods normalized by this maximum on the horizontal axis. The quantities plotted on the vertical axes are the arithmetic mean of graph instance completion times, the standard deviation among the completion times, and the % application processor (AP) efficiency.

The instance completion times are normalized with respect to the input arrival period. Thus, the normalized instance completion time should be unity on ideal machines that meet the application requirements for any input rate. Ideally, a value greater than unity indicates that the machine cannot meet the application rates. However, in this case, due to a finite sized window of observation of the application behavior, the average values plotted are approximately unity when the machine meets the application requirements.

The plots of standard deviation between the instance completion times give a better idea of the non-uniformity in execution. The input period is used to normalize the difference between the completion time of an instance and the ideal completion time. The % efficiency indicates the time for which the execution unit at the application processor was busy performing useful computation and not waiting for data.

4.1 Correlator Application

This graph was chosen to represent a simple, yet realistic, signal processing application. The corresponding graph appears in Fig. 8 [Tec90b]. The circles indicate the nodes to be executed and the arrows represent the queues holding the data required by the nodes. "T" represents the threshold value required before the destination node becomes *ready*. "R" represents the amount that is read by the destination node on execution setup. "C" represents the amount that is consumed on destination node breakdown. "P" represents the production amount from the previous node. Actual execution times for the primitives listed beside the nodes were obtained from the signal processing primitives library [Tec90a]. It was simulated assuming five processors and five memory modules.

The points obtained for the graphs plotted in the case of the correlator graph were taken at 5%intervals except in the region of close similarity where the interval was 1%. The results for normalized mean are shown in Fig. 9 and 10. While the difference between FCFS and RC is not discernible in Fig. 9, Fig. 10 clearly indicates that the RC algorithm reaches unity 5% before the FCFS algorithm. At all times the RC curve remains below the FCFS curve on the graph. The normalized standard deviation, shown in Fig. 11, indicates that the RC algorithm provides a more uniform output than does the FCFS algorithm throughout the range of input data periods. Due to the dependencies inserted by the RC algorithm, the processor efficiency is lower for the RC case than for the FCFS case until uniformity in output is obtained as shown in Fig 12. This result is caused by the dependencies inhibiting the earlier nodes in the graph from executing until they are satisfied. While the efficiency is slightly lower for the RC approach, the lower normalized mean and standard deviation results indicate an improvement by use of the RC algorithm over the FCFS scheduling technique.

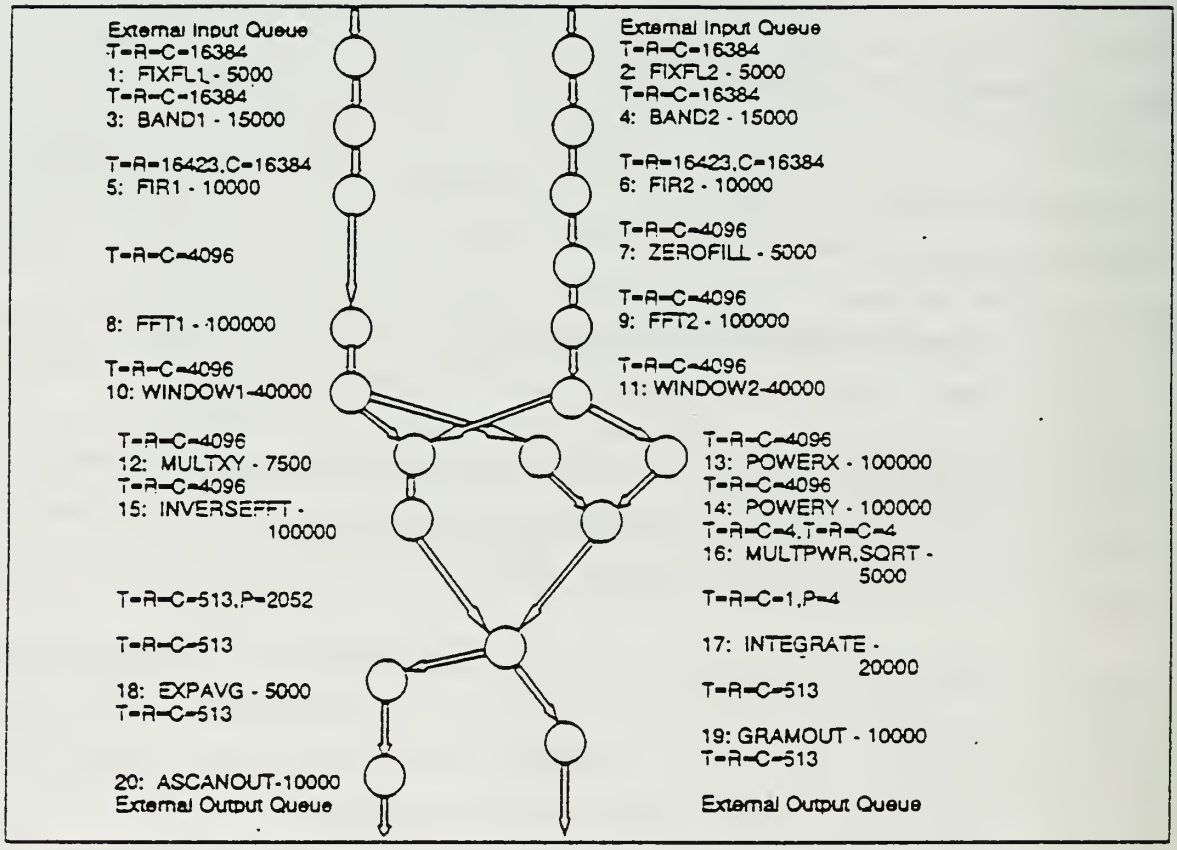

Figure 8: Data-flow Graph for the Correlator Application

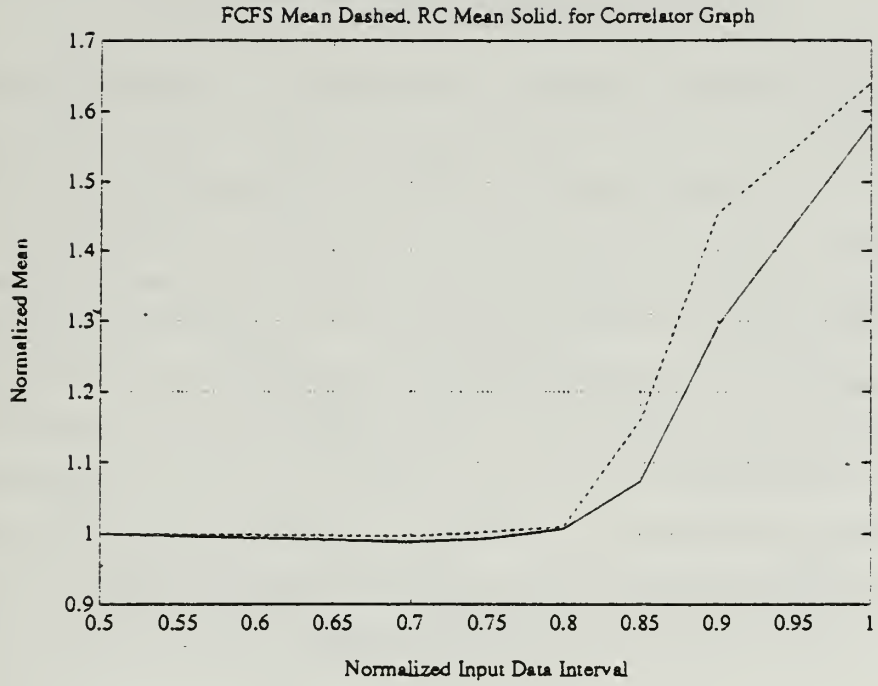

Figure 9: Correlator Graph - Mean Instance Completion Times

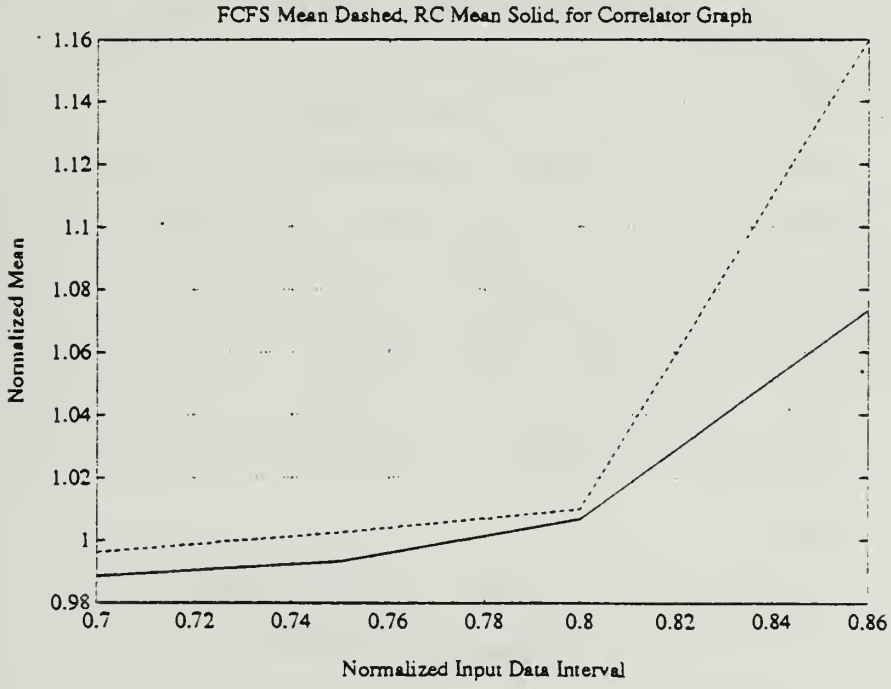

Figure 10: Correlator Graph - Blow-up of Mean Instance Completion Times

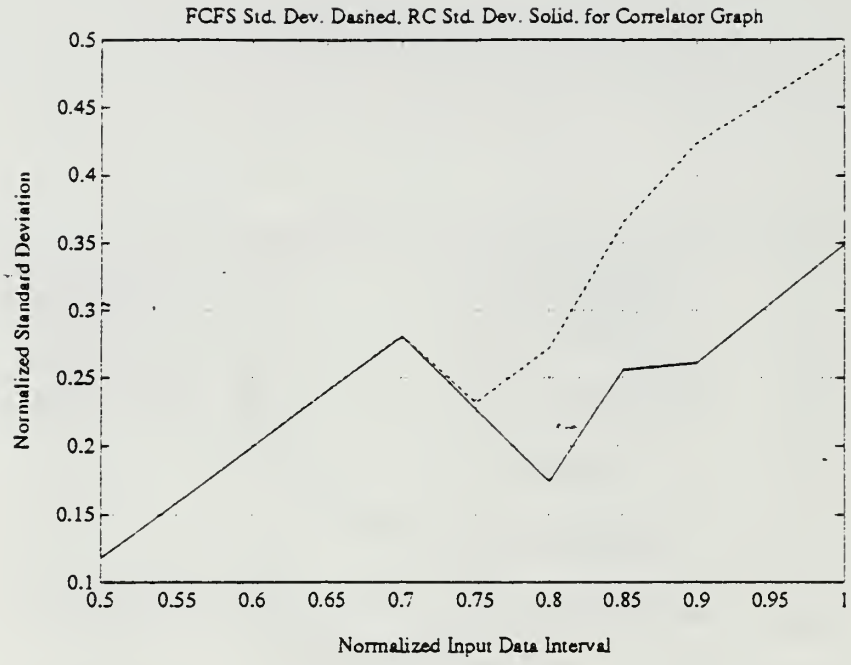

Figure 11: Correlator Graph - Standard Deviation

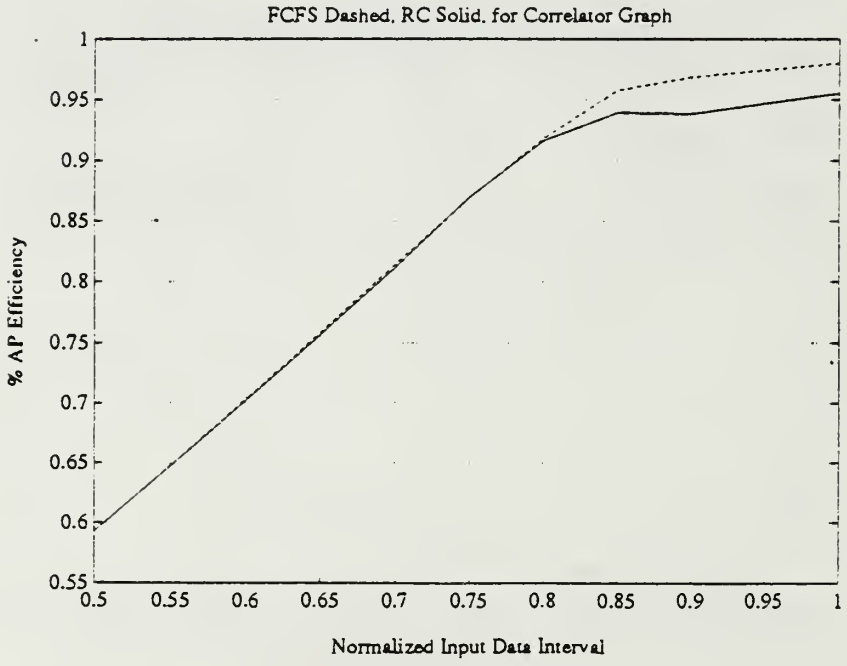

Figure 12: Correlator Graph - Processor Efficiency

4.2 Fast Fourier Transform Data-flow Graph

The fast Fourier Transform (FFT) algorithm was chosen to examine the effects of the RC analysis on a communication intensive graph. The graph for a 2-D FFT can be represented in terms of that of a one dimensional (1-D) FFT. This application assumes a 256 point vector of inputs. The 1-D FFT can be calculated in log 256 stages of operations with 128 operations per stage. Each stage can be divided into p parallel tasks, with $\frac{256}{2p}$ operations per task. As the tasks in stage i finish, they send their outputs to the tasks in stage $i+1$. The data-flow graph for a 2-D FFT uses 2 log 256 stages to transform a 256 x 256 matrix of inputs. 256 1-D FFT's are computed for rows followed by another 256 1-D FFT's for columns. Tasks in the first 8 stages perform 1-D FFT's on all 256 rows with each task performing $\frac{256}{2p}$ operations. Tasks in stage log 256 send data to tasks in stage $(8+1)$ in such a way that the second set of 8 stages performs 256 column transforms. The numbers beside the queues represent queue over threshold, production, and consume values in micro-seconds. The 2-D FFT graph is shown in Fig. 13.

This data-flow graph was simulated on a machine with 8 processors and 8 memory modules. The normalized mean for FFT is shown in Figs. 14 and 15. Here also. the input data rate is met 5% before that of the FCFS algorithm when RC-based restructuring is used. Due to the high communication overhead as compared to the previous graph, the input rate met satisfied by this machine is lower. The normalized standard deviations are shown in Figs. 16 and 17. Again, clearly the RC standard deviation outperforms the FCFS standard deviation throughout the spectrum of input data rates. The normalized standard deviation is consistently less than 0.5 regardless of load level. Figure 18 demonstrates the differences in processor efficiency for the FFT graph. The low values are caused by the communication overhead involved in processing this type of graph. The restructured graph yields a greater processor efficiency due to the assigned dependencies limiting the data movement traffic. This implies that a much more uniform output results from the RC algorithm regardless of load.

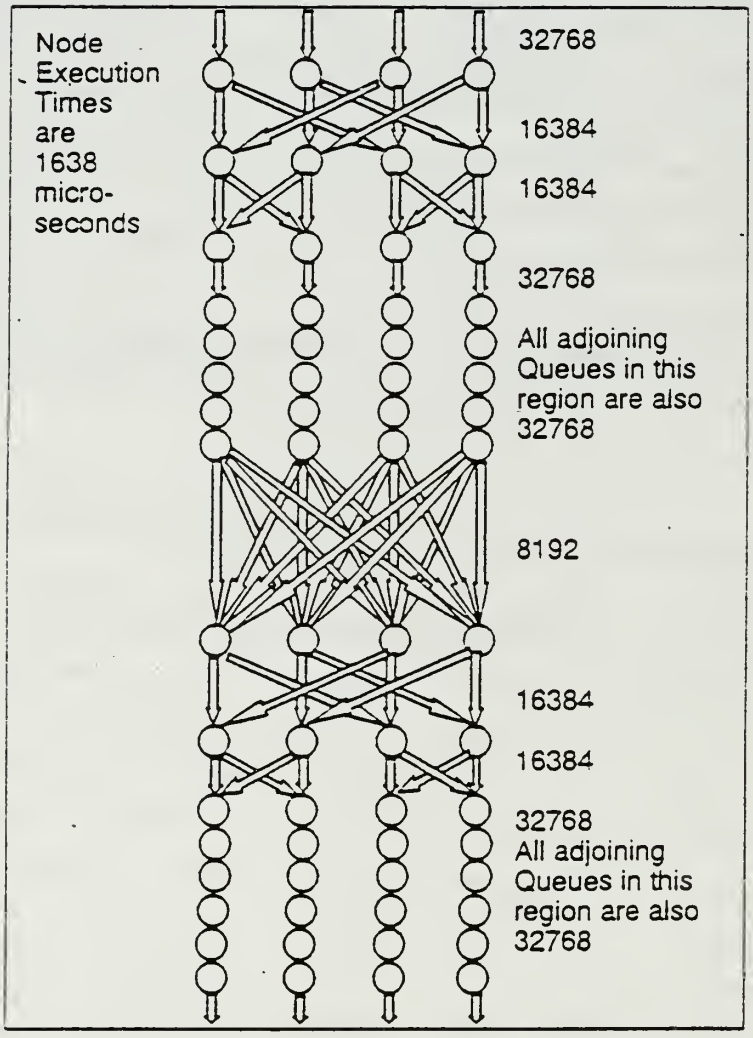

Figure 13: 2-D FFT Data-flow Graph

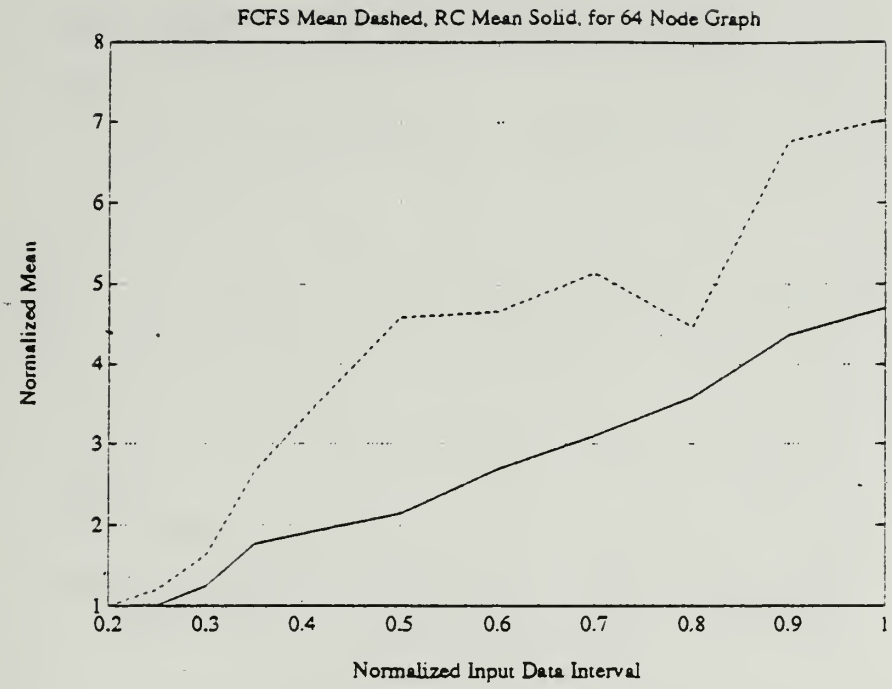

Figure 14: FFT Graph - Mean Instance Completion Times

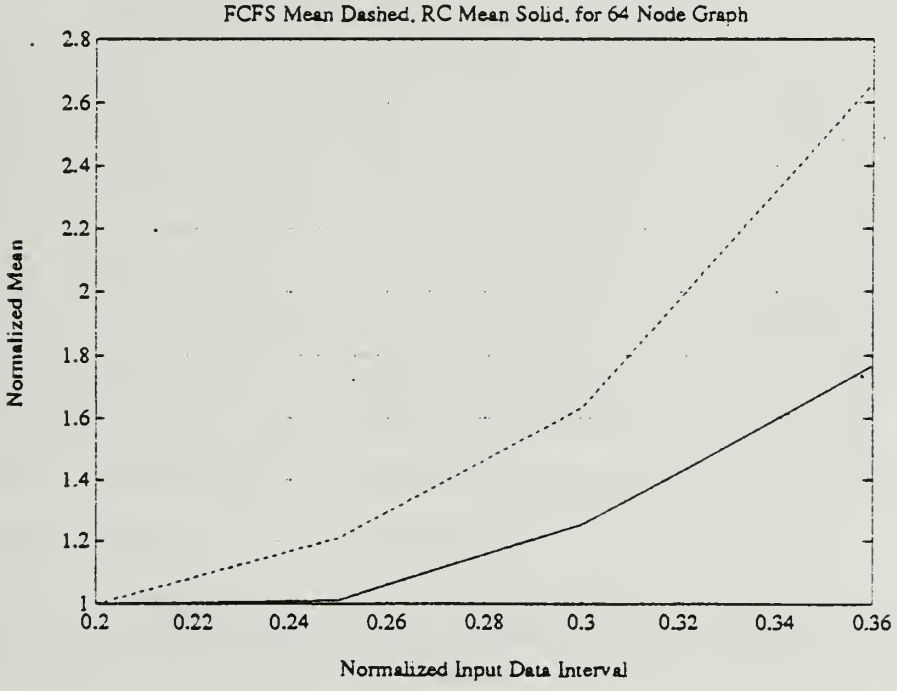

Figure 15: FFT Graph - Blow-up of Mean Instance Completion Times

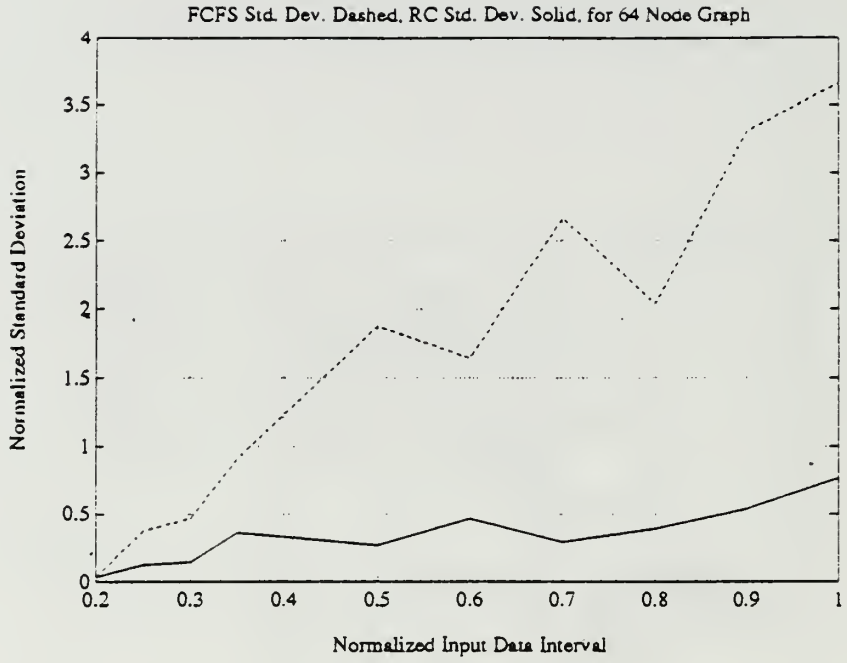

Figure 16: FFT Graph - Standard Deviation

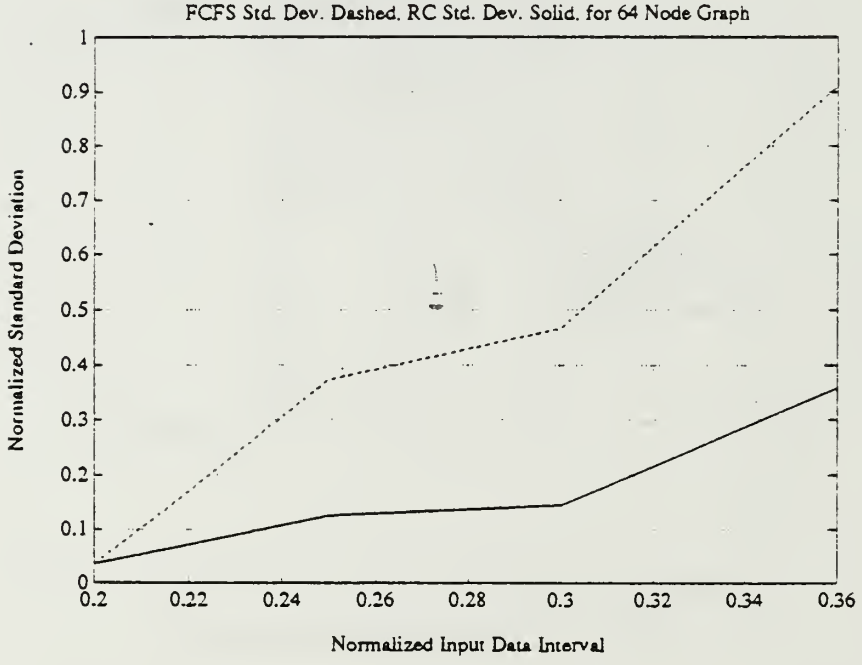

Figure 17: FFT Graph - Blow-up of Standard Deviation

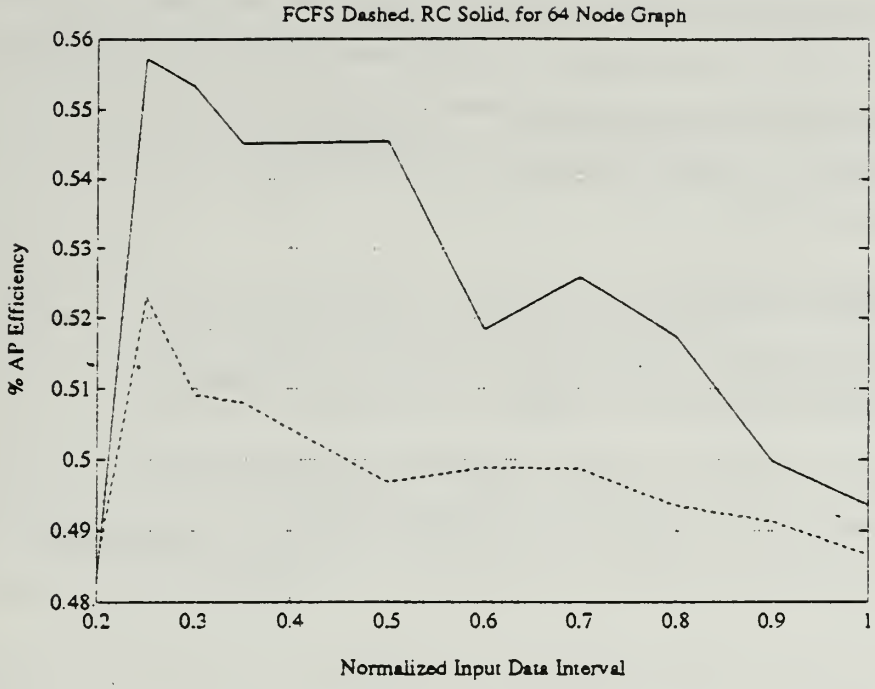

Figure 18: FFT Graph - Processor Efficiency

5 Concluding Remarks and Future Research

In conclusion, the major contribution of this work has been to present a compile-time approach to the enable efficient use of the data-flow paradigm in real-time applications with periodic arrival of data. We have shown that the proposed approach of RC analysis provides a framework in which optimizations related to data-flow execution at the task-level can be carried out. In order to control the execution when input data arrives periodically, this technique restructures the application graph that has a more predictable behavior under the same run-time mechanism. The results have been presented using typical applications, *viz.*, the correlator and FFT graphs. They show that this approach does make the individual instance completion time more uniform regardless of the the input period and the communication overhead.

Currently, the following issues with regard to the use of compile-time data-flow graph analysis are being investigated.

- Chaining of nodes results in saving the breakdown and setup overhead. However, unrestrained chaining results in loss of parallelism and could be detrimental to processor efficiency. It is difficult to predict the effect of chaining two nodes for a FCFS execution; but if chaining is specified within the framework of RC analysis, its effect can be accurately predicted.

- Given a specific assignment, it is known which queues are accessed at the same time. This information can be used to algorithmically assign memory modules to queues, so that the interference between nodes at a module is minimized.

- There are several ways in which the additional dependencies can be introduced. The criteria to select the minimal set of dependencies to be introduced that provide the minimal, yet effective, control of the execution are being developed.

References

[Bro87] S. A. Brobst. Organization of an instruction scheduling and token storage unit in a tagged token data flow machine. In *Proceedings of the 1987 International Conference on Parallel Processing*, volume 3. August 1987.

[GKW85] J. R. Gurd, C. C. Kirkhame, and I. Watson. The Manchester Prototype Dataflow Computer. *Communications of the ACM*, January 1985.

[KCN90] C.-T. King, W.-H. Chou, and L. M. Ni. Pipelined data-parallel algorithms: Part i—concept and modeling. *IEEE Transactions on Parallel and Distributed Systems*, October 1990.

[KM66] R. M. Karp and R. E. Miller. Properties of a model for parallel computations: Determinacy, termination, queueing. *SIAM Journal of Applied Mathematics*, 14(6). November 1966.

[Kog81] P. M. Kogge. *The Architecture of Pipeline Computers*. McGraw-Hill, 1981.

[LB90] E. A. Lee and J. C. Bier. Architectures for statically scheduled dataflow. *Journal of Parallel and Distributed Computing*, 10:333–348, December 1990.

[Lee91] E. A. Lee. Consistency in dataflow graphs. *IEEE Transactions on Parallel and Distributed Systems*, 2(2), April 1991.

[LM87] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous dataflow programs for digital signal processing. *IEEE Transactions on Computers*, C-36(1). January 1987.

[RGP82] B. R. Rau, C. D. Glaeser, and R. L. Picard. Efficient code generation for horizontal architectures: Compiler technique and architectural support. In *Proceedings of the 9th International Symposium on Computer Architecture*, 1982.

[Ric90] M. L. Rice. Navy's new standard digital signal processor: The AN/UYS-2. In *Proceedings of the Association of Scientists and Engineers 27th Annual Technical Symposium*, May 1990.

[SA91] S. B. Shukla and D. P. Agrawal. Scheduling pipelined communication in distributed memory multiprocessors for real-time applications. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, May 1991.

[SFP83] P. S. Sawkar, T. J. Forquer, and R. P. Perry. Programmable modular signal processor - a data flow computer system for real time signal processing. In *Proceedings of the 1983 International Conference on Parallel Processing*, volume 3. August 1983.

[SMS90] S. Som, R. R. Mielke, and J. W. Stoughton. Strategies for predictability in real-time data-flow architectures. In *Real-time Systems Symposium*, pages 226–237, 1990.

[Tec90a] AT&T Technologies. AN/UYS-2 graph primitives library - floating point. Technical Report 5885404, AT&T Bell Laboratories, 1990.

[Tec90b] AT&T Technologies. ECOS workstation user manual. Technical Report Alpha 890301-01, AT&T Bell Laboratories, 1990.

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2.	Library, Code 52 Naval Postgraduate School Monterey, CA 93943-5002	2
3.	Lieutenant Commander Steve Kasputis Department of the Navy Naval Sea Systems Command (PMS 412) Washington, DC 20362-5101	1
4.	Mr. Richard Stevens Commander of Naval Research Laboratory 4555 Overlook Avenue S. W. Washington, DC 20375-5000	1
5.	Mr. Jerome L. Uhrig, WH 46243 AT&T Bell Laboratories 67 Whippany Road P. O. Box 903 Whippany, NJ 07981-0903	1
6.	Chairman, Code EC Department of Electrical & Computer Engineering Naval Postgraduate School Monterey, CA 93943-5004	1
7.	Prof. Shridhar B. Shukla, Code EC/Sh Department of Electrical & Computer Engineering Naval Postgraduate School Monterey, CA 93943-5004	1
8.	Prof. Amr Zaky, Code CS/Za Computer Science Department Naval Postgraduate School Monterey, CA 93943-5004	1

9. Mr. Al Miller
TRW Inc.
Suite 800
1735 Jefferson Davis Highway
Arlington, VA 22202 1

10. Research Office
Code 08
Naval Postgraduate School 1